Instrumenting LogP Parameters in GM: Implementation and Validation

Edgar A. León, Arthur B. Maccabe
Scalable Systems Lab
Computer Science Department
The University of New Mexico
Albuquerque, NM 87131-1386
leon,maccabe @cs.unm.edu

Ron Brightwell
Scalable Computing Systems Department
Sandia National Laboratories
Org 9223, MS 1110
Albuquerque, NM 87185-1110
bright@cs.sandia.gov

Abstract

This paper describes an apparatus which can be used to vary communication performance parameters for MPI applications, and provides a tool to analyze the impact of communication performance on parallel applications. Our apparatus is based on Myrinet (along with GM). We use an extension of the LogP model to allow higher flexibility in determining the parameter(s) to which parallel applications may be more sensitive to. We show that individual communication parameters can be controlled within a small percentage error, and that the other parameters remain unchanged.

1 Introduction

Parallel architectures are driven by the needs of applications. Application developers and users, often deal with a variety of problems regarding the performance of applications on specific parallel architectures. The origin of these is diverse and may be related to different aspects of the application and the platform, such as scalability, latency and bandwidth issues, balance between communication and computation, etc. We have created an apparatus to identify the causes of these problems regarding communication performance, and thus help users and developers in improving overall application performance and to make decisions on which parallel architectures may or may not be suitable for their applications.

In this paper, we describe a tool to vary communication parameters on high-performance clusters. This tool should prove useful in analyzing the requirements and sensitivity of applications to communication performance. Our apparatus is based on the LogP model, which has been useful in characterizing communication performance and extended in different ways, to provide specific characterizations in terms of message size, type of network, etc. We also extend this model to specify a more fine grain characterization of the network, providing greater detail in the sensitivity of applications to network performance.

Over the last few years, Myrinet has become the high-performance clusters interconnect of preference over the community. Together with Myrinet, the GM message-passing system has been widely used and, in many cases, has served as the basis for customized communication systems that use Myrinet. In this work, we have instrumented an extension of the LogP communication parameters into GM, so we can arbitrarily vary communication performance. Despite the fact that we implement our apparatus in GM, we do not expect applications to be written in GM to benefit from it, but in MPI (Message Passing Interface), which defines a standard, portable message-passing system. MPI has been ported to a variety of platforms, in particular to Myrinet through GM. Thus, any library implemented on top of GM will provide the apparatus.

Our apparatus allows a better understanding of parallel applications and may suggest possible communication bottlenecks that would otherwise be hard to detect. It is also specific in terms of which communication parameter(s) the application may be more sensitive to. It also provides application designers with a better understanding of the architectural requirements of their applications, and if a certain platform (or upgrade) may or may not improve the performance of their applications in a significant way.

The remainder of this paper is organized as follows. Sections 2 and 3 give a brief overview of GM and MPICH-GM respectively. A description of the communication parameters used in our apparatus is presented in Section 4. Sec-

This work was supported in part by Sandia National Laboratories under contract number AP-1739.

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 IEEE

tion 5 describes the instrumentation of the LogP parameters in GM. Section 6 describes our measurement methodology and shows the results of our empirical validation and calibration of the communication parameters. A discussion of related work is presented in Section 7. Finally, Section 8 presents our conclusions and outlines our intentions for future work.

2 Glenn's Messages (GM)

GM [12] is a low-level message-passing system created by Myricom[1] for its Myrinet networks [3]. GM is comprised of a driver, a library, and a Myrinet Control Program (MCP) that runs on the network interface. A Myrinet interface in turn, is comprised of the LANai [13] and local SRAM memory.

GM provides reliable, in order delivery of messages. It allows any non-privileged user to use the network interface directly without the intervention of the host Operating System (OS). This technique, called OS-bypass, offloads part of the OS functionality to the network interface, avoiding the need to interrupt the OS to handle message sends or receives. GM has several internal queues to handle the communication between the LANai and the user. These queues reside in either, host virtual memory or LANai SRAM.

A message to be sent over the network is divided in fragments called packets. Although Myrinet does not impose any limitation on the size of the fragment to send, GM uses packets of GM_MTU (= 4KB) length, where it reaches its maximum bandwidth.

Flow control in GM is done at two levels. At the user level, GM provides send and receive tokens to the user. A send token is needed by the user to send a message over the network. Similarly, a receive token is needed in order to post a receive buffer. When the operation completes, the token is passed back to the user. The number of tokens (GM_NUM_SEND_TOKENS + GM_NUM_RECV-_TOKENS) assigned to a user determines how many slots there are to use in certain LANai queues and it is based on a page size of the host machine. At the MCP level, GM has an "ACK/NACK-based go back N" flow control protocol.

The MCP is a state based program and controls the transactions on its four state machines: SDMA, SEND, RDMA and RECV, see Figure 1. Depending on the state of the "system", the MCP will trigger events for the different state machine interfaces to control the flow of messages between the network and the user, and vice-versa.

Upon reception of a packet, the RECV state machine checks its header, if it's not valid, the packet is dropped. If the header is valid, it DMAs the packet to a buffer or staging area in LANai SRAM (at this point the packet is being called a chunk). The RECV machine notifies the RDMA machine about chunks in the SRAM receive buffers.

The RDMA state machine is in charge of dealing with the message fragments or chunks in LANai SRAM. Among other tasks, the RDMA machine looks up for user receive buffers in the receive queue that matches the message size and priority of the incoming message. If the message is bigger than a constant (GM_MAX_FAST_RECV_BYTES = 128) and there's no matching buffer, the message is dropped. Otherwise, the message is DMAed to the matching user buffer and an event describing the receive is enqueued in a receive event queue in host memory.

The user is in charge of polling the receive event queue to check for receive events (and other events) that have completed. The user does that by using the function `gm_receive()` (part of the library interface) and other similar functions[2].

The user sends a message by invoking the GM library function `gm_send_with_callback()` and other similar send functions[3]. This function passes the send descriptor (or token) to the send queue in LANai SRAM. Note that prior to the send, the user should allocate the message in DMAable memory at the host (this can be done by using the library functions `gm_malloc()` or `gm_register_memory()`), to ensure that the DMA engine would be able to transfer the message directly from user space to LANai SRAM.

The SDMA state machine removes an entry from the send queue, and begins the transfer of the message to LANai SRAM in chunks that fit in the send buffers, as the buffers become available. It will also notify the SEND state machine to begin the delivery of the chunks. The SEND machine then injects the chunks as packets to the network prepending the correct route to the destination node, and records the send to the sent list.

3 MPI on GM (MPICH-GM)

MPICH-GM is a port of MPICH [6], a portable implementation of MPI, to GM. MPICH-GM is a two-level protocol, it uses an *eager* protocol for the transmission of small messages, and a *rendez-vous* protocol for long messages. The two-level protocol reflects the trade-offs to achieve low-latency and high-bandwidth.

The eager protocol allows the transmission of small messages, even when a receive buffer has not been posted by the receiver. The receiver temporarily stores the incoming message until the message is consumed. This technique allows low-latency but low-bandwidth due to the extra copy at the receive side. This protocol is non-blocking since it allows

[1] http://www.myri.com.

[2] `gm_blocking_receive()` and `gm_blocking_receive_no_spin()`.

[3] `gm_send_to_peer_with_callback()` and `gm_directed_send_with-_callback()`

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 IEEE

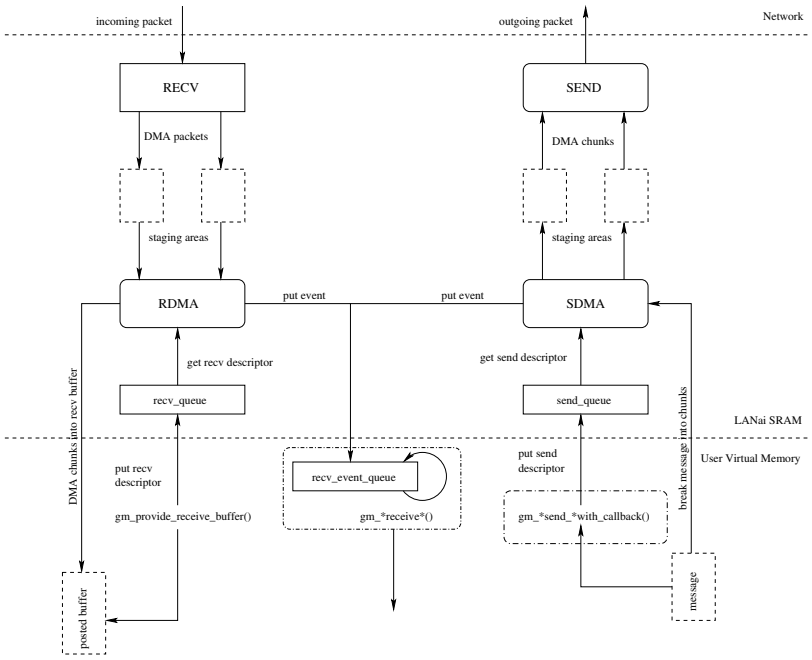

Figure 1. The GM MCP and its interaction with user space in a message transmission and reception. Dash rectangles refer to buffer space, square rectangles denominate either a state machine or a queue. Rounded rectangles, identify the areas where the $Lo_so_rg_sg_rGP$ parameters are implemented.

the sender to complete even when there's no matching receive.

To avoid significant overhead in memory copies for long messages, the rendez-vous protocol implements a 3-way handshaking. The sender transmits a request-to-send to the receiver, the receiver replies back to the sender with a clear-to-send, and finally the sender transmits the messages. The reply from the receiver contains the virtual memory address in which the message should be delivered, thus the sender performs a remote put operation to move the message to its destination. In GM, the remote put operation is implemented via the function `gm_directed_send_with_callback()`. Thus, this protocol achieves higher bandwidth but lower latency due to the handshake.

An interesting feature of MPICH-GM is the ability to change the behavior of blocking receive MPI calls. Three modes are provided: *polling*, *blocking* and *hybrid*. By default it uses the polling method (through the GM function `gm_receive()`) in which the network devices is polled until an event is found. This method achieves low latency, but high CPU utilization. In the blocking method, the MPI function sleeps in the kernel. Upon each event, the network interface delivers an interrupt to the host to awaken the sleeping function. This behavior is achieved through the GM function `gm_blocking_receive_no_spin()`. This method allows low CPU utilization, but the interrupt and context switch costs increase the latency. In the hybrid method, the MPI function polls for 1 millisecond and then goes to sleep. This method uses the GM function `gm_blocking_receive()`.

4 The $Lo_so_rg_sg_rGP$ parameters

Our tool is based on an extension to the *LogP* communication parameters [4]. LogP is a model for distributed-memory multiprocessors and abstracts the parallel architecture into four parameters:

- **L**atency: an upper bound on the time to transmit a small message.
- **o**verhead: the time that the host processor is engaged in sending or receiving a message and cannot do any other work.
- **g**ap: the minimum time interval between consecutive message transmissions or consecutive message receptions at a node. The reciprocal of g corresponds to the available per-node bandwidth
- **P**rocessors: the number of nodes in the system.

This model is asynchronous and the latency may vary (although bounded by the parameter L). It considers that the

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 **IEEE**

network has a finite capacity, i.e., at most L/g messages may be on the network at any given time.

The model does not differentiate between short and long messages, thus it does not account for special devices to support the transmission of long messages. To address this issue, an extension of this model was created: *LogGP* model [1]. The new parameter, G, defines the time per byte for a long message. As in the LogP model, the reciprocal of G characterizes the available per processor communication bandwidth for long messages.

Under this model, the time to transmit a small message from one process to another in different nodes, takes: o L o cycles. To transmit a long message of k bytes, takes: o $(k-1)G$ L o. First, the sender processor initiates the transfer incurring in o, subsequent bytes are sent every G cycles. The last byte enters the network at time o $(k-1)G$ and arrives at the host L o cycles later.

Note that the send and receive overhead are not distinguished. Considering that the send and receive operations are usually not symmetric, we use two independent parameters to model the overhead: the send overhead, o_s, and the receive overhead, o_r.

Another issue arises while considering the instrumentation of the gap. The gap is the minimum time interval between consecutive sends or consecutive receives; if we only control the gap on the send side, in a many-to-one communication semantics, the receiver may be receiving consecutive messages faster than the gap; if we only control the receive side, in a one-to-many communication semantics, the sender may send messages faster than the gap. Thus, we consider the gap as being composed of two parameters: the send gap, g_s, and the receive gap, g_r. We consider these two separately and independent of each other.

In summary, we characterize the communication performance of a parallel machine in terms of seven parameters: L, o_s, o_r, g_s, g_r, G and P. These parameters allow us a high-degree of flexibility to determine the particular parameter(s) to which an application may be sensitive to.

5 Implementation

We instrument the communication parameters based on GM version 1.2.3. Timing measurements are done at two levels: (1) at the MCP level, using the Real-Time-Clock (RTC) in the Myrinet interface, and (2) at the host user level, using the function `gettimeofday()`. The Myrinet interfaces used in this work have an RTC reference period of $\frac{1}{2}\mu s$.

5.1 Latency (L)

GM maintains a queue of events in host memory (receive event queue) which is filled by the MCP when an event occurs. The GM user, through the library, polls this queue for new events such as the completion of a receive. The queue is accessed through a port.

To increase the latency without modifying other LogP parameters, we added a delay queue as shown in Figure 2. The delay queue has the same size as the receive event queue and it is accessed through the same port. When an event is inserted in the event queue, the time at which the event should be delivered (time of arrival + delay) is inserted into the delay queue. When the user polls for events, the delay queue is polled for events ready to be delivered.

From the LogP model,

$$\frac{RTT}{2} = o_s \quad L \quad o_r, \tag{1}$$

$$L = \frac{RTT}{2} - o_s - o_r \tag{2}$$

In a message exchange, the delay parameter is set in both parties to the desired added delay value x and the formula for the new latency L becomes:

$$\begin{aligned} L &= \frac{RTT}{2} - o_s - o_r && \text{by (2)} \\ &= \frac{RTT \quad 2x}{2} - o_s - o_r \\ &= \frac{RTT}{2} - o_s - o_r \quad x \\ &= L \quad x && \text{by (2)} \end{aligned} \tag{3}$$

To implement the added delay, we modified the implementations of two library functions: `gm_open()` and `gm_receive()`. In `gm_open()` we set the delay value and allocate and initialize the delay queue. The delay itself was added in `gm_receive()`. This function returns an event if there is an event in the event queue or no event if the queue is empty. When a new event is inserted in the event queue, the function checks if the event is a receive event, if so, the delayed delivery time is calculated and inserted in the delay queue. Once the new time is calculated, the delay queue is polled for events ready to be delivered. When there are no events pending in the event queue, the delay queue is polled to check for events ready to be delivered.

An important issue arises due to the `gm_unknown()` library function. The user may pass events to this function to be handled. For example, the user may only be interested in RECV_EVENT (normal receive event), passing all other events to `gm_unknown()`. If a FAST_RECV_EVENT (receive event for small messages) arises, `gm_unknown()` will convert this event into a RECV_EVENT. To perform the conversion, `gm_unknown()` replaces the type of the event and rewinds the current event queue pointer to the previous slot. To avoid adding delay latency twice for an event such as a FAST_RECV_EVENT that was passed

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 IEEE

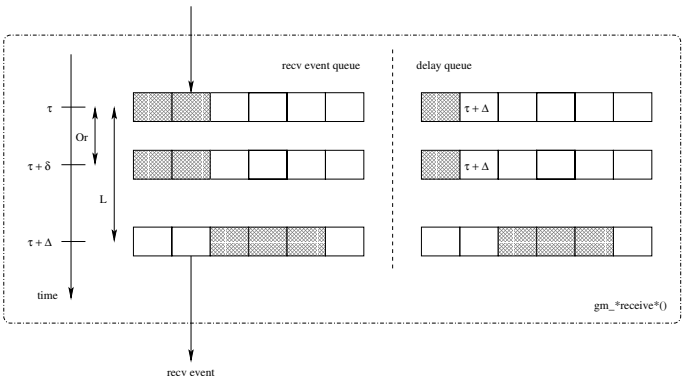

Figure 2. Implementation of L and o_r. A new event is inserted in the event queue at time τ, o_r has been increased by δ and L has been increased by Δ. If $\Delta = 0$, the event will be delivered at time τ δ.

to `gm_unknown()`, the delay queue implementation checks for `gm_unknown()` calls by detecting if the current receive queue pointer gets re-wound, and if so, just delivers the event to the application without computing the new delay time.

5.2 Overhead (o_s o_r)

In contrast to latency, varying receive overhead requires that we take the processor away from the application for the prescribed period of time. The receive overhead was implemented by adding the following delay loop into `gm_receive()`:

```
while (time_ready_to_deliver > current_time)
  get_current_time(&current_time)
```

The send overhead was implemented by adding a similar delay loop in the function `gm_*send_*with_callback()` after the user has initiated the send (`gm_send_with_callback()`, `gm_send_to_peer_with_callback()` or `gm_directed_send_with_callback()`) and before the transfer of the message to the LANai SRAM has been initiated, see Figure 3.

Modifying the send and receive overhead in this fashion allows latency to remain unaffected. Consider adding a delay of x μs to o_s (in both parties) in a message exchange:

$$
\begin{aligned}
L &= \frac{RTT}{2} - o_s - o_r && \text{by (2)} \\
&= \frac{RTT \quad 2x}{2} - (o_s \quad x) - o_r && \text{thus,} \\
&= \frac{RTT}{2} - o_s - o_r \\
&= L && \text{by (2)}
\end{aligned}
$$

A similar argument shows that increasing the receive overhead is independent of the latency. The increase in the overhead is not independent of the gap. When increasing the send (or receive) overhead, the gap increases as well because the time between consecutive message sends (receives) will be increased by the extra time to execute every send (receive).

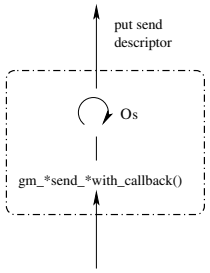

Figure 3. Implementation of o_s. Before the message gets written to LANai SRAM, the user is delayed by o_s.

5.3 Gap per message (g_s g_r)

The gap, or minimum time interval between consecutive sends or receives, can be more specifically described as a parameter composed of two sub-parameters: send gap and receive gap. The send gap was implemented by enforcing a delay between the sending of consecutive messages. Similarly, the receive gap enforces a delay between the delivery of consecutive messages.

To implement the receive gap (send gap), we modified the RDMA (SDMA) state machines. In particular, we introduced the following delay enforcement between the receipt (delivery) of a message and the delivery of the corresponding event:

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 **IEEE**

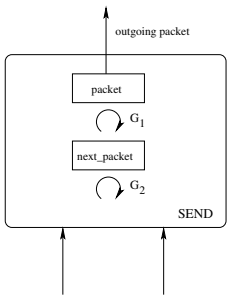

Figure 4. Implementation of G. After every packet send and before the next send, delay the SEND state machine by G. Note that G depends on the size of the packet (G_1 for packet and G_2 for next_packet.

```
// time reference is 1/2 us.
while (RTC < ready);
ready = RTC + 2*gap_delay
Deliver or Send
```

5.4 Gap per byte for long messages (G)

The Gap is implemented by adding a delay, G, after every byte sent. To achieve this, we first calculate the number of bytes in every packet to be sent. Then, we add a delay, according to the length of the packet, after the packet has been sent. If the length of the packet is n bytes, then the delay is:

$$n \quad G = G \tag{4}$$

G can be calculated using the LogP signature for bursts of bulk messages. If a message exceed 256 bytes, meaning that a packet exceed 256 bytes, then it is considered a bulk message and the SEND state machine is delayed after the packet was sent, see Figure 4. It is interesting to note that GM partition the message data in similar chunks, each fitting in one packet without exceeding the minimum number of packets that the message comprises. For example if 4097 bytes are sent, then one packet will have 2048 bytes and the other 2049 bytes, instead of 4096 and 1 respectively.

In order to calculate G, we use the LogP signature for bursts of bulk messages. At the steady state, we calculate G as the average time to send a message (just as in the LogP model for g). Thus,

$$BW = \frac{1}{G} \tag{5}$$

$$BW = \frac{n}{G} \qquad \text{by (4)} \tag{6}$$

We increase the size of the message, n, up to a point where the bandwidth does not increase anymore. This point is exactly the packet size, GM_MTU = 4096 bytes.

The implementation of the Gap, has to take into account the fact that the MCP does not support division operations, thus we used bit shifting.

The delay of a packet of size n is:

$$G = (n \quad x)\mu s = (n/2^x)\mu s \qquad \text{thus,}$$

$$BW = \frac{n}{G} = \frac{nbytes}{\frac{n}{2^x}\mu s} = 2^x \text{MB/s}$$

Therefore, a delay of key x, would modified the bandwidth to 2^x MB/s.

The implementation of this parameter resides in the SEND state machine in the MCP. Here's the pseudo code:

```
// sml=send-message limit,
// smp=send-message pointer
// these vars are not the actual registers
 packet_length = sml - smp - header_size
// RTC=Real-Time Clock
 if (packet_length > 256)
   while (RTC < ready_to_send);

// send the packet by writing to register SMLT
// (Send-Message Limit, with the Tail)
 SMLT = sml;

// (time reference is equal to 1/2 us.)
 if (packet_length > 256)
   ready_to_send = RTC + 2*(packet_length >> x)
```

6 Validation and Calibration

Each node contained a 500 MHz Intel Pentium III processor with 256 MB of main memory and a Myrinet LANai 7.2 network interface card with 2 MB of memory. Nodes were connected using a Myrinet 8-port SAN switch. Our results were gathered using GM version 1.2.3 and a Linux 2.2.14 kernel.

6.1 Measurement Methodology

To extract the communication parameters of a given parallel machine, we use a micro-benchmark based on [5]. It was implemented on GM and it is based on the Myricom's `logp_test` program. A brief description of this micro-benchmark follows.

The micro-benchmark issues a sequence of M request messages of a fixed length, and measures the average time per issue, *message cost*. The receiving node sends a reply (of the same length as the request) to the sender for every message received. Here's the sender's pseudo code:

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 IEEE

```
start timer
  repeat M times
    issue request
stop timer
  ... handle remaining replies
```

For small M (initial phase) no replies are handled (and hopefully the network has not reached its capacity), therefore the message cost is only the send overhead, o_s. For bigger M, the sender begins to receive replies (transition phase) and therefore the message cost increases due to the receive overhead, o_r. This cost keeps increasing up the point where the network has reached its capacity (steady phase). At this point, the sender cannot inject one message, before draining a reply from the network. Thus, after sending a message, the sender waits a period of time, *idle*, then receives a reply from the network, and then sends the next message. Therefore, the gap, g, is comprised of:

$$g = o_s \quad idle \quad o_r \tag{7}$$

which is just the time interval between consecutive sends.

As Figure 5 shows, three stages or phases can be identified. In the initial phase the message cost is just o_s. The transition phase begins either at the reception of the first reply or when the network capacity has been reached, whichever comes first. The reception of the first reply should occur after the Round-Trip-Time, RTT, of the first request, i.e., after the transmission of $\frac{RTT}{o_s}$ number of messages. The steady phase is reached when the network is full and the message cost is just the gap.

In GM the transition phase is reached earlier than expected (M = GM_NUM_SEND_TOKENS), due to the limitation on the number of sends imposed by the tokens mechanism.

It is easy to measure g and o_s using the signature graph shown in Figure 5: g is measured as the average message cost at the steady phase; o_s is measured as the average message cost in the initial phase. To calculate o_r from equation 7, it is necessary to determine the value of *idle*. Since this value is not known, a delay Δ between message sends is added. As Figure 5 shows, for $\Delta \quad idle$, equation 7 becomes:

$$g = o_s \quad \Delta \quad o_r \tag{8}$$

Having o_r, is easy to calculate the latency, L, from:

$$\frac{RTT}{2} = o_s \quad L \quad o_r \tag{9}$$

RTT is easily measured by taking the average of a number (100) of ping-pong trials with the same message size used in the micro-benchmark.

The resulting micro-benchmark follows:

```
start timer
  repeat M times
    issue request
    compute for Delta time
stop timer
  ... handle remaining replies
```

The parameters measured depend on the message size used by this micro-benchmark; for example, to get the latency, we use a small message size as defined in the LogP model. However, we can also measure the "latency" for bulk messages although not defined in the original model. Thus, the parameters L, o_s, o_r and g are functions of the message size n: $L(n)$, $o_s(n)$, $o_r(n)$ and $g(n)$.

To measure G (gap per byte), we first measure $g(k)$, where k is the number of bytes at which the maximum bandwidth is reached (4KB in GM); then, G is calculated using equation 4 where $G = g(k)$ (see section 5.4). Therefore, with this micro-benchmark, and a ping-pong test to get the round-trip-time, we are able to obtain the values of the LogP parameters.

6.2 Results

To empirically validate and calibrate the LogP parameters, we vary each one of them to a fixed-value letting the remaining parameters to be unmodified. By using the micro-benchmark described in the previous section, we measure the value of all the parameters and verify that the desired value of the varied parameter is (in average) no more than 9% different from the observed value. We also verify that the remaining parameters remain constant and with low standard deviation (std).

Tables 1, 2 and 3 show the results of varying L, o_s and o_r respectively for messages of size 8 bytes (see Sections 5.1 and 5.2). The In label represents the desired added value of the varied parameter to the system, for example if the inherent system's $o_s = 1.34$, then an added value of 20 gives us a desired value of 21.34. The *Out* label represents the measured values using the micro-benchmark. The arbitrary increase on the overhead in Tables 2 and 3 is not independent of the gap. When increasing the send overhead, the gap increases as well because the time between consecutive message sends will be increased by the extra time to execute every send. The same argument is true for receive overhead.

Tables 4 and 5 show the results of varying g_s and g_r respectively for messages of size 8 bytes (see Section 5.3). The $In^{=}$ label represents the desired value of the varied parameter, naturally a desired value less than the inherent system's value cannot be achieved. The values in column RTT_1 differ from RTT in the number of trials of the ping-pong test: 1 in the former, and 100 in the latter. The reason of having RTT_1 for these tables is to avoid the appearance of gap in the round-trip-time and so the latency should remain

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 IEEE

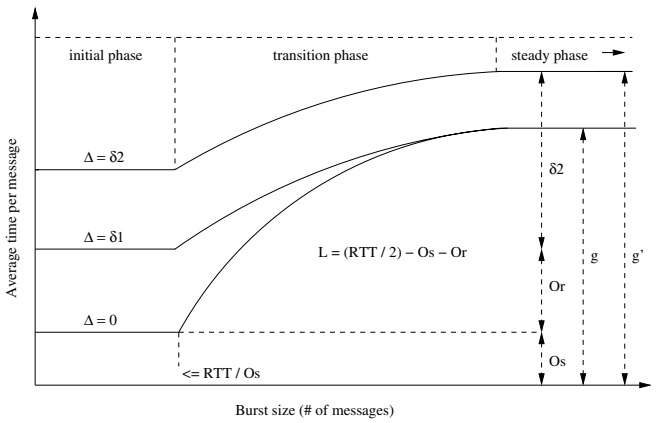

Figure 5. Expected micro-benchmark signature of the LogP parameters.

Table 1. Varying L for messages of length 8 bytes.

	In	Out				
%err	L	L	o_s	o_r	g	RTT
	0	18.72	1.38	4.31	23.91	48.84
6.90	10	29.41	1.39	4.28	24.11	70.19
5.60	20	39.84	1.39	4.30	24.03	91.08
9.53	30	51.58	1.39	4.29	23.94	114.54
9.38	40	62.47	1.39	4.30	24.19	136.33
5.16	50	71.30	1.36	4.36	24.06	154.07
7.13	60	83.00	1.34	4.35	24.13	177.40
6.99	70	93.61	1.39	4.30	23.90	198.64
5.94	80	103.47	1.39	5.01	23.94	219.77
6.63	90	114.69	1.33	4.37	23.86	240.81
6.17	100	124.89	1.40	4.28	24.00	261.16
6.32	110	135.67	1.39	4.31	23.94	282.76
5.74	120	145.61	1.40	4.30	23.97	302.64
avg 6.79			1.38	4.36	23.99	
std 1.37			0.02	0.19	0.09	

Table 2. Varying o_s for messages of length 8 bytes.

	In	Out				
%err	o_s	o_s	o_r	g	RTT	L
	0	1.34	4.33	23.92	48.82	18.73
2.40	10	11.58	4.12	25.39	68.42	18.49
0.15	20	21.31	3.82	31.22	88.72	19.21
0.57	30	31.51	4.18	41.39	108.65	18.62
3.50	40	42.74	3.07	51.29	128.48	18.41
0.02	50	51.33	4.19	61.23	151.54	20.24
0.43	60	61.60	4.09	71.21	169.05	18.83
0.16	70	71.23	4.17	81.19	191.97	20.57
0.00	80	81.34	4.34	91.26	212.04	20.32
0.33	90	91.04	4.67	101.28	232.01	20.28
0.01	100	101.35	4.18	111.24	252.01	20.46
0.09	110	111.44	4.35	121.27	271.38	19.90
0.13	120	121.18	4.63	131.14	290.76	19.55
avg 0.64			4.16			19.50
std 1.11			0.39			0.82

Table 3. Varying o_r for messages of length 8 bytes.

	In	Out				
%err	o_r	o_r	o_s	g	RTT	L
	0	4.29	1.38	24.20	48.82	18.73
2.10	10	14.08	1.40	25.19	68.39	18.71
2.10	20	23.87	1.38	27.62	89.00	19.24
3.17	30	33.34	1.38	36.64	108.77	19.66
2.83	40	43.16	1.41	45.84	129.11	19.97
2.50	50	53.04	1.41	55.25	149.09	20.09
2.88	60	62.56	1.41	64.49	168.52	20.27
2.59	70	72.48	1.35	73.67	191.41	21.87
3.96	80	81.12	1.40	82.83	212.28	23.61
3.20	90	91.41	1.36	92.29	232.47	23.45
3.39	100	100.90	1.38	101.66	252.20	23.81
0.73	110	113.49	1.37	110.83	271.51	20.87
2.24	120	121.60	1.41	120.13	292.47	23.21
avg 2.64			1.38			21.03
std 0.82			0.02			1.91

constant. Since just one trial is considered, the value of the latency is bigger than in RTT columns due to "warm-up issues".

Table 6 shows the results of varying G for messages of size 4088 bytes (see Section 5.4). Column κ represents the input key values, in which a system with key x has a bandwidth (BW) of 2^x MB/s. The RTT is not independent of G, because of the size of the message.

The units in which the parameters are measured are: $L(\mu s)$, $o_s(\mu s)$, $o_r(\mu s)$, $g_s(\mu s)$, $g_r(\mu s)$, $G\,(\mu s)$, RTT(μs), BW(MB/s). The *%err* label in the tables represents the percentage error difference between the desired value, v_i, and the observed value, v_o, and it is calculated as follows:

$$\%err = \frac{v_i - v_o}{v} \quad 100 \tag{10}$$

where v is the desired or added value (column In), depending on the table.

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 **IEEE**

Table 6. Varying G for messages of length 4088 bytes.

	In=			Out					
	%err	κ	BW	G	G	o_s	o_r	BW	RTT	L
		7	128	31.93	76.29	1.55	3.27	53.57	290.06	140.21
		6	64	63.87	77.22	1.53	4.00	52.93	290.23	139.57
	9.22	5	32	127.75	115.97	1.57	4.21	35.24	291.65	140.03
	9.38	4	16	255.50	231.54	1.53	4.09	17.65	304.25	146.49
	9.47	3	8	511.00	462.62	1.59	3.80	8.83	514.17	251.68
	8.01	2	4	1022.00	940.16	1.58	3.92	4.34	1033.59	511.28
	6.94	1	2	2044.00	1902.16	1.60	3.88	2.14	2088.05	1038.53
avg	8.60					1.56	3.88			
std	1.10					0.02	0.30			

Table 4. Varying g_s for messages of length 8 bytes.

	In=		Out				
	%err	g_s	g	o_s	o_r	RTT_1	L
		0	23.91	1.36	4.33	125.07	56.84
		10	24.18	1.36	4.30	125.06	56.85
		20	27.70	1.35	4.33	125.06	56.84
	10.83	30	33.25	1.41	4.25	126.06	57.36
	1.95	40	40.78	1.38	4.28	125.06	56.86
	0.72	50	50.36	1.40	4.26	124.07	56.36
	0.52	60	60.31	1.52	4.15	124.06	56.35
	0.17	70	70.12	1.50	4.17	124.06	56.34
	0.23	80	80.18	1.39	4.27	125.05	56.85
	0.20	90	90.18	1.36	4.33	126.06	57.33
	0.29	100	100.29	1.41	4.27	125.06	56.85
	0.21	110	110.23	1.40	4.31	125.06	56.81
	0.15	120	120.18	1.37	4.37	125.06	56.78
avg	1.52			1.40	4.27	124.98	56.80
std	3.31			0.05	0.06	0.63	0.31

Table 5. Varying g_r for messages of length 8 bytes.

	In=		Out				
	%err	g_r	g	o_s	o_r	RTT_1	L
		0	24.13	1.36	4.32	125.06	56.84
		10	23.92	1.39	4.29	126.06	57.34
		20	23.79	1.38	4.94	125.06	56.19
	1.27	30	29.62	1.41	4.28	125.06	56.83
	2.43	40	39.03	1.37	4.31	125.06	56.84
	3.08	50	48.46	1.41	4.26	125.06	56.85
	3.30	60	58.02	1.39	4.29	125.06	56.84
	3.31	70	67.68	1.41	4.24	126.06	57.37
	3.33	80	77.34	1.39	4.29	125.06	56.84
	3.33	90	87.00	1.39	4.28	125.06	56.85
	3.34	100	96.66	1.40	4.27	125.06	56.84
	3.33	110	106.34	1.41	4.26	126.06	57.35
	3.26	120	116.09	1.37	4.31	127.06	57.84
avg	2.99			1.39	4.33	125.44	56.98
std	0.66			0.01	0.18	0.65	0.40

7 Related Work

This work provides an apparatus to instrument LogP communication parameters in high-performance commodity clusters. We consider the send and receive overhead separately as well as the gap (o_s, o_r, g_s, g_r), to provide greater flexibility in analyzing the sensitivity of applications to each one independently. We use a micro-benchmark based on [5] to measure the communication parameters. The related work can be classified in two groups: (1) work focused on the sensitivity of applications to certain communication parameters, and (2) work focused on the analysis and measurement of communication parameters.

The first group is comprised by [7, 11, 2, 9] and their goals are similar in spirit to ours, but in different contexts. In [7], the authors study the performance of network interface controllers in cache coherent DSM (Distributed Shared Memory) machines. They found that the controller *occupancy* is critical to the performance of these machines. The occupancy is the time in which the controller is busy with one action and cannot perform another, and partially corresponds to the latency and gap in the LogP model.

In [11], the authors study the impact of communication parameters on parallel applications. The applications they used were written in Split-C on top of Generic Active Messages. They found the host overhead to be critical to application performance. Our work uses GM as the communication layer and will focus on applications written in MPI, although could be used by any library that uses GM underneath.

In [2], the authors examine the impact of communication parameters in SVM (Shared Virtual Memory) systems. They found that the overhead of generating and delivering interrupts is critical in a SVM system. In [9], the authors artificially increase latency and bandwidth simulating communication performance of WANs (Wide Area Networks), to analyze collective communication operations on this type of network.

The second group comprised by [8, 9, 10] provide differ-

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 **IEEE**

ent micro-benchmarks to measure communication parameters based on the LogP model. In [8], the authors consider the send and receive overhead separately, and the latency, overhead and gap for bulk data transfers are dependent on the message size. They use two micro-benchmarks similar to [5], one measures the o_s and g, and the other, o_s o_r. The goal of this study is to analyze the performance of the FM (Fast Messages) library in different platforms.

In [9], the authors introduce another extension to the LogP model, the *parameterized LogP*, suitable for WANs. In this model, the overhead and gap are also considered as functions of the message size, and the overhead is also split in send and receive overhead. The latency is defined as an end-to-end latency from process to process, which differs from the original model. This work presents improved algorithms for collective communication on WANs.

In [10], the authors provide a micro-benchmark to measure the parameterized LogP. The aim of this micro-benchmark is the ability to measure the communication parameters without saturating the communication links (except for measuring the gap for a message of size 0), i.e., use fewer messages than in [5]. This micro-benchmark called *MPI LogP benchmark* was implemented in MPI.

8 Conclusions and Future Work

This work provides an apparatus to vary communication performance based on the LogP model on high-performance commodity clusters. It is intended to help parallel application developers and users to identify the source of performance degradation of an application on parallel architectures.

We have empirically calibrated and validated our apparatus to show that we can control the communication parameters within a percentage error of no more than 9%. We have also shown that the parameters can be varied independently of each other.

We are going to extend our apparatus to allow the variation of communication parameters in terms of functions that may simulate events such as interrupt coalescing, slow start and others. Also, we plan to use our apparatus in several MPI applications available from Sandia National Labs and others, to determine their sensitivity to communication performance.

References

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model – one step closer towards a realistic model for parallel computation. In *7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '95)*, pages 95–105, June 1995.

[2] A. Bilas and J. P. Singh. The effects of communication parameters on end performance of shared virtual memory clusters. In *Proceedings of the ACM/IEEE SuperComputing (SC '97) Conference*, San Jose, CA, Nov. 1997.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[4] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '99)*, pages 1–12, San Diego, CA, May 1993.

[5] D. E. Culler, L. T. Liu, R. P. Martin, and C. O. Yoshikawa. Assessing fast network interfaces. *IEEE Micro*, 16(1):35–43, 1996.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[7] C. Holt, M. Heinrich, J. P. Singh, E. Rothberg, and J. Hennessy. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Stanford University, Jan. 1995.

[8] G. Iannello, M. Lauria, and S. Mercolino. Cross-platform analysis of Fast Messages for Myrinet. In D. K. Panda and C. B. Stunkel, editors, *Proceedings of the 2nd International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications (CANPC '98)*, volume 1362 of *Lecture Notes in Computer Science*, pages 217–231, Las Vegas, NV, Feb. 1998. Springer.

[9] T. Kielmann, H. E. Bal, and S. Gorlatch. Bandwidth-efficient collective communication for clustered wide area systems. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '00) Conference*, Cancun, Mexico, May 2000.

[10] T. Kielmann, H. E. Bal, and K. Verstoep. Fast measurement of LogP parameters for message passing platforms. In *4th Workshop on Runtime Systems for Parallel Programming (RTSPP '00)*, Cancun, Mexico, May 2000.

[11] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA '97)*, pages 85–97, Denver, CO, June 1997.

[12] Myricom, Inc. *The GM API*, Oct. 1999. `http://www.myri.com/scs/GM/doc/gm_toc.html`.

[13] Myricom, Inc. *LANai 7*, June 1999. `http://www.myri.com/vlsi/LANai7.pdf`.

Proceedings of the 27th Annual IEEE Conference on Local Computer Networks (LCN'02)
0742-1303/02 $17.00 © 2002 **IEEE**